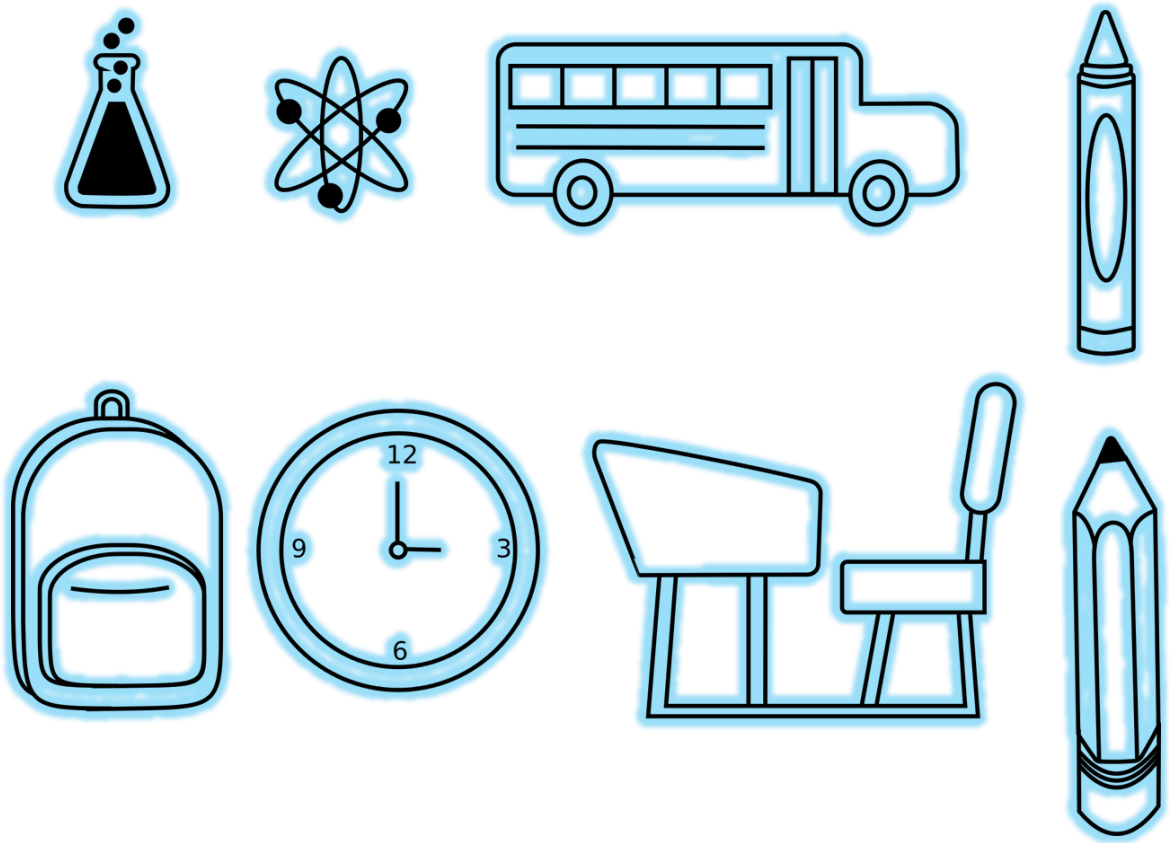
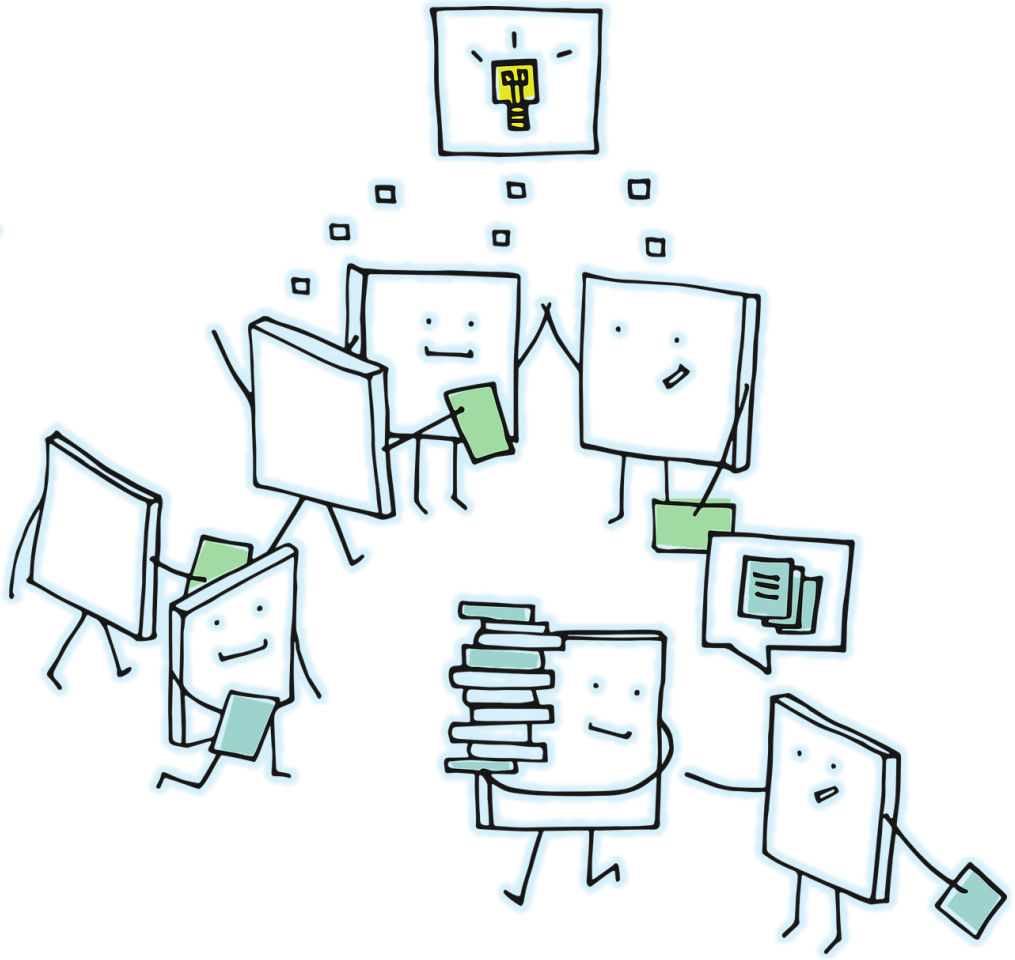
Programación Orientada a Objetos

Primer contacto



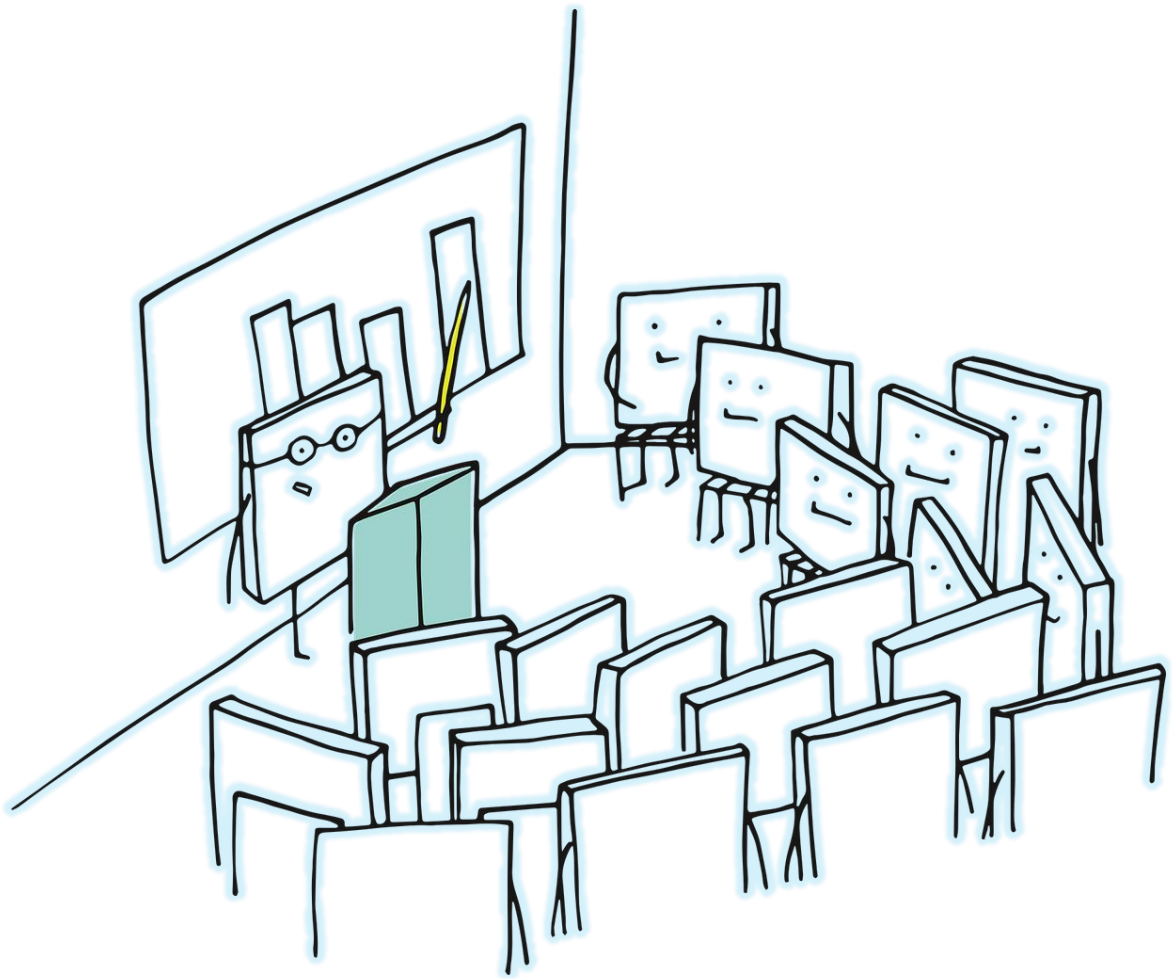
La Programación orientada a objectos nos viene acompañando desde hace más de seis décadas, y lo seguirá haciendo durante mucho tiempo más.

Su explosión se produjo en la década de los 80 y 90 del siglo pasado, con su adopción por parte de lenguajes como C++ y Java.



En el mundo real solemos distinguir entre seres animados e inanimados, soliendo referirnos a estos últimos como objetos.

Cuando en el mundo del desarrollo de software usamos el paradigma de la programación orientada a objetos, y aunque en el mundo real suene mal, cualquier cosa o entidad que queramos representar, puede considerarse un objeto, incluyendo personas y animales.



Ya que vamos a comenzar una formación de Programación orientada a objetos (POO), vamos a usar una escuela como ejemplo.

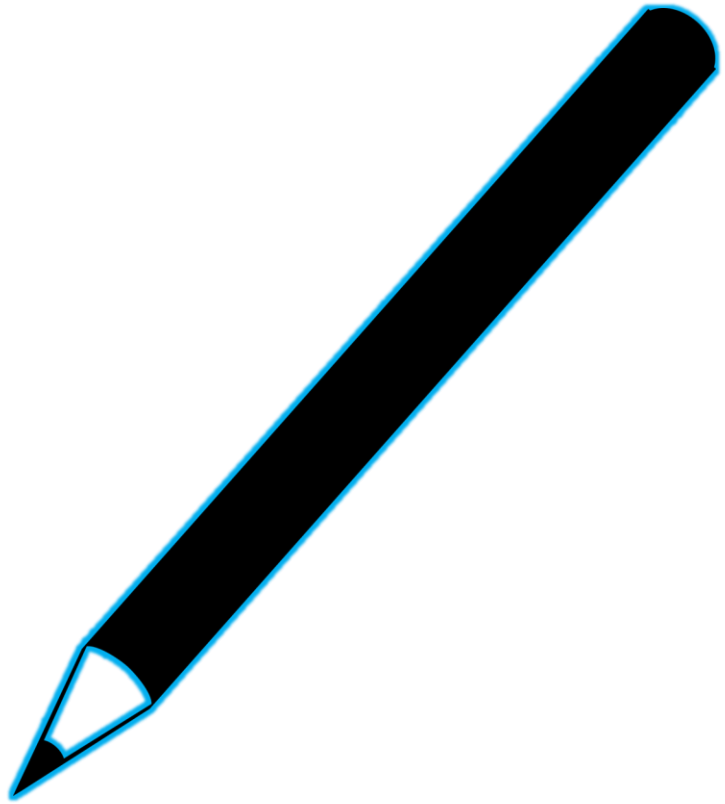
Si nos encargaran desarrollar una aplicación para gestionar una escuela en un lenguaje orientado a objetos, deberíamos empezar por identificar con que clases de objetos nos podríamos encontrar.

Por ejemplo:

- Material escolar (libros, lápices, pizarras, proyectores …)
- Alumnos/as
- Profesores/as
- Personal de limpieza
- Aulas, bibliotecas, sillas, pupitres
- Titulaciones, cursos, asignaturas, evaluaciones …
- Y muchas más cosas dependiendo del alcance que tenga que tener la aplicación a desarrollar.

Aunque podrían ser muchas las **clases de objetos** que necesitaríamos para gestionar una escuela, nos centraremos en unas pocas para ilustrar aquellos conceptos que necesitamos conocer sobre este paradigma de programación.

Así pues, utilizaremos, lápices, alumnos/as, profesores/as, y un autobús escolar para para fijar las ideas principales que nos aportaran luz sobre la POO.



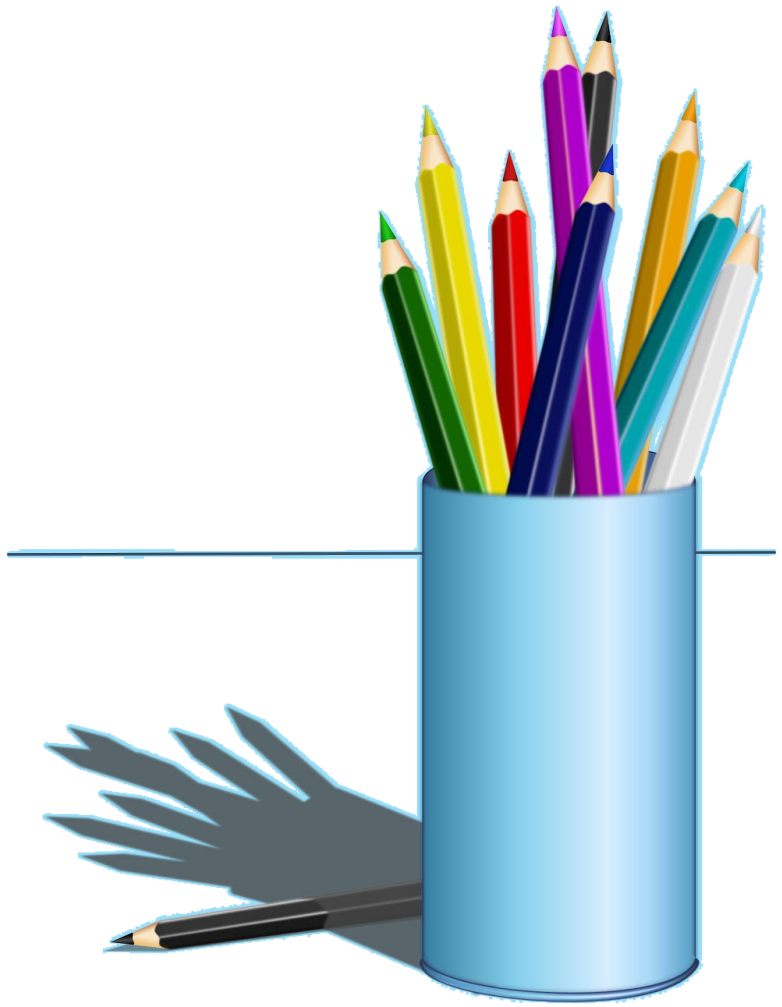
Clase de objeto **Lápiz**

En POO, cualquier **clase de objeto** consta de dos grandes grupos de miembros:

- Propiedades o características
- Métodos (acciones)

Con las propiedades de un lápiz, podríamos describir el grosor de la punta, el material del que está hecho, el color, …

Los métodos son aquellas acciones que esta clase de objeto puede llegar a realizar, como por ejemplo dibujar, y si incorpora goma, borrar.



Los objetos **Lápiz**

Fijaos que, hasta ahora, no hemos estado hablando de objetos, sino de clases de objetos.

Y sí, son cosas diferentes:

- Una clase de objetos, viene a ser como un molde o una fábrica, que define que propiedades y métodos tendrán los objetos concretos que se creen a partir de ella.

- Los objetos generados a partir de una clase, llamados instancias, son los lápices concretos en que sus propiedades estarán dotadas de valores (grosor 1, color rojo, material madera), y que podrán accionar sus métodos, como dibujar y borrar.

```java
class Lapiz {

    float grosor;
    String material;
    String color;

    void dibujar(int x, int y, char contenido) {
        // Escribiremos con
        // el grosor y color
        // establecidos en
        // las posiciones
        // x e y.
    }

    void borrar(int x, int y) {
        // Borraremos el
        // contenido escrito
        // en las posiciones
        // x e y.
    }

}
```

La clase **Lápiz**

En este ejemplo codificado en Java, podemos observar como declararíamos la clase de objeto **lápiz**, con las tres propiedades y los dos métodos usados anteriormente como ejemplo.

class es la palabra reservada de java que indica que Lápiz será una matriz o fábrica de objetos de ese tipo (lápices).

float se utiliza para declarar variables numéricas con decimales.

int sirve para declarar variables numéricas de tipo entero.

char, con el que podemos escribir cualquier carácter.

String se usa para declarar variables de tipo alfanumérico.

void indica que los métodos dibujar y borrar no devuelven nada.

```
Lapiz lapiz01 = new Lapiz();
lapiz01.color = "rojo";
lapiz01.grosor = 0.2f;
lapiz01.material = "madera";
lapiz01.dibujar(1, 10, 'H');
lapiz01.dibujar(2, 10, 'o');
lapiz01.dibujar(3, 10, 'l');
lapiz01.dibujar(4, 10, 'a');
lapiz01.borrar(20, 11);


Lapiz lapiz02 = new Lapiz();
lapiz02.color = "azul";
lapiz02.grosor = 0.3f;
lapiz02.material = "plástico";
lapiz02.dibujar(6, 10, 'M');
lapiz02.dibujar(7, 10, 'a');
lapiz02.dibujar(8, 10, 's');
lapiz02.dibujar(9, 10, 't');
lapiz02.dibujar(10, 10, 'e');
lapiz02.dibujar(11, 10, 'r');
lapiz02.dibujar(12, 10, 's');
lapiz02.borrar(21, 10);
```

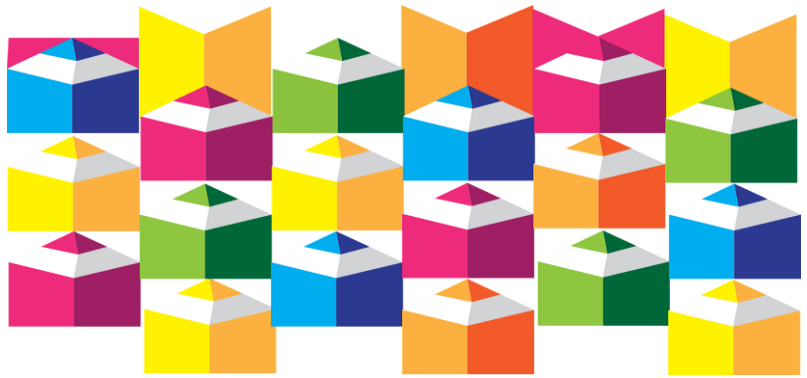
Hola **Masters**

Las instancias (objetos concretos) de la clase **Lápiz**

Una vez definida una clase de objetos, en nuestro caso Lápiz, podremos obtener tantas instancias (Lapiz01, Lapiz02, …) de la clase Lápiz como necesitemos.

Para ello, en Java se usa la palabra reservada new junto al constructor (concepto que veremos conforme avance el curso) de su tipo de clase.

Para asignar o leer propiedades, o llamar a métodos de una instancia, se hace intercalando un punto entre el nombre de la instancia y la propiedad o método a llamar.



Pilares de la POO

Herencia, encapsulación, abstracción, y polimorfismo.

Los cuatro pilares de la POO: **La Herencia**

Vamos a cambiar de ejemplo, y vamos a referirnos ahora a algunas de las personas que pueden integrar una escuela, como son alumnos/as y profesores/as.

Si pensamos en el alumnado, podríamos decidir que los datos (propiedades) que necesitamos saber son la fecha de nacimiento, el DNI, su nombre, apellidos, su dirección, y su nota de acceso.

Por otro lado, supongamos que del profesorado necesitamos saber la fecha de nacimiento, el DNI, su nombre, apellidos, su dirección, y su titulación.

Como vemos, excepto por una propiedad, alumnado y profesorado comparten el resto de propiedades.

Los cuatro pilares de la POO: **La Herencia**

```
class Alumno {

    String dni;
    String nombre;
    String apellidos;
    String direccion;
    LocalDate fechaNacimiento;
    float notaAcceso;

}
```

```
class Profesor {

    String dni;
    String nombre;
    String apellidos;
    String direccion;
    LocalDate fechaNacimiento;
    String titulacion;
}
```

Una primera alternativa de codificación, incluir todas las propiedades directamente en las clases Profesor y Alumno, tratándolas como clases de objetos independientes.

Si además de alumnos y profesores, también tuviéramos que gestionar personal de dirección, personal de limpieza, vigilantes, etc., probablemente deberíamos repetir en cada una de estas clases de objetos, las propiedades DNI, nombre, apellidos, dirección, y fecha de nacimiento.

Probablemente ya, nos estemos dando cuenta de que necesitaríamos algún tipo de mecanismo que nos permitiera definir una sola vez aquellas propiedades o métodos compartidas entre clases de objetos similares, evitando así repetir tanto código.

Los cuatro pilares de la POO: **La Herencia**

```
class Persona {

    String dni;
    String nombre;
    String apellidos;
    String direccion;
    LocalDate fechaNacimiento;
}
```

```
class Alumno extends Persona {

    float notaAcceso;

}
```

```
class Profesor extends Persona {

    String titulacion;
}
```

Y efectivamente, ese mecanismo existe, y es la **herencia**.

Como vemos, hemos generalizado las clases Alumno y Profesor en una nueva clase llamada Persona, que incluye aquellas propiedades que Alumno y Profesor tienen en común, quedándose únicamente aquellas que les son específicas, como nota de acceso y titulación.

En el caso de Java, la manera de codificar que una clase herede de otra, es usando la palabra reservada extends, seguida del nombre de la clase de la que queremos heredar.

Con este mecanismo, escribimos las propiedades o métodos comunes una sola vez en la clase base, pero a efectos prácticos, las propiedades de la clase Persona se consideran incluidas en todas sus clases derivadas (Alumno y Profesor).

Los cuatro pilares de la POO: **La Herencia**

```
Alumno alumno01 = new Alumno();
alumno01.dni = "11111111A";
alumno01.apellidos = "García Jimenez";
alumno01.nombre = "Ana";
alumno01.direccion = "C/Balmes, 10 - 08010 Barcelona";
alumno01.fechaNacimiento = LocalDate.of(2000, 03, 18);
alumno01.notaAcceso = 9.87f;

Profesor profesor01 = new Profesor();
profesor01.dni = "22222222B";
profesor01.apellidos = "Lopez Gomez";
profesor01.nombre = "Rosa";
profesor01.direccion = "C/Roger de flor, 23 - 08012 Barcelona";
profesor01.fechaNacimiento = LocalDate.of(1990, 12, 07);
profesor01.titulacion = "Administración y Dirección de Empresas";
```

Como podemos comprobar, a la hora de instanciar y utilizar objetos concretos de la clase Alumno o Profesor, podemos acceder a aquellas propiedades que no incluyen directamente, pero que heredan de su clase base Persona y, que en definitiva, acaban formando parte de ellas.

Y como no podía ser de otra manera, también tienen acceso a aquellas propiedades que tienen definidas directamente, como son la nota de acceso y la titulación.

Los cuatro pilares de la POO: **Encapsulación**

```java
class Persona {

    private String dni;

    public void asignaDni(String dniRecibido) {
        // Validaríamos si el DNI recibido
        // es correcto, y solamente en caso
        // de que sí lo fuera, lo asignaríamos
        // a la propiedad dni.

        dni = dniRecibido;
    }

    String nombre;
    String apellidos;
    String direccion;
    LocalDate fechaNacimiento;
}
```

```java
Alumno alumno01 = new Alumno();
alumno01.asignaDni("11111111A");
alumno01.apellidos = "García Jimenez";
alumno01.nombre = "Ana";
alumno01.direccion = "C/Balmes, 10 - 08010 Barcelona";
alumno01.fechaNacimiento = LocalDate.of(2000, 03, 18);
alumno01.notaAcceso = 9.87f;

Profesor profesor01 = new Profesor();
profesor01.asignaDni("22222222B");
profesor01.apellidos = "Lopez Gomez";
profesor01.nombre = "Rosa";
profesor01.direccion = "C/Roger de flor, 23 - 08012 Barcelona";
profesor01.fechaNacimiento = LocalDate.of(1990, 12, 07);
profesor01.titulacion = "Administración y Dirección de Empresas";
```

La encapsulación hace referencia a la posibilidad de ocultar ciertas propiedades o métodos a clases externas.

Por ejemplo, la letra final del DNI depende de un cálculo en función del resto de números.

Si permitimos que se pueda asignar el DNI directamente a la propiedad, corremos el riesgo de que nos llegue un DNI inválido.

Si ocultamos dicha propiedad al exterior (añadiendo el modificador de acceso private), y proveemos un método de asignación (con el modificador de acceso public), donde verifiquemos en base a ese cálculo, que es un DNI correcto, evitaremos la asignación de un DNI inválido.

Los cuatro pilares de la POO: **Abstracción**

La abstracción suele usarse en combinación con la encapsulación, pero hace referencia a un concepto diferente.

El objetivo de la abstracción es ocultar las complejidades del interior de las clases, ofreciendo a cambio una interfaz más abstracta y general.

Supongamos un autobús escolar, en el que el conductor maneja el vehículo a través de los mandos del salpicadero y los pedales.

Acciones como poner el intermitente, acelerar, frenar, etc., se basan en esta interfaz (salpicadero, pedales, …) que abstraen al conductor de las complejidades del sistema eléctrico y mecánico, y el conjunto de acciones internas que conllevan que la luz se encienda, o que el coche incremente su velocidad o se frente.

Los cuatro pilares de la POO: **Abstracción**

```java
class BusSchool {

    public void acelerar() {
        acelerarFase01();
        acelerarFase02();
        acelerarFase03();
    }

    public void frenar() {
        frenarFase01();
        frenarFase02();
    }

    public void activarIntermitenteDerecho() {

        activarIntermitenteFase01();
        activarIntermitenteFase02();
    }

    public void desactivarIntermitenteDerecho() {

    }

    private void acelerarFase01() {
        // ...
    }

    private void acelerarFase02() {
        // ...
    }

    private void acelerarFase03() {
        // ...
    }

    private void frenarFase01() {
        // ...
```

En este ejemplo codificado en Java, podemos ver como los métodos acelerar(), frenar(), y activarIntermitenteDerecho(), son los que abstraen a los usuarios que instancian la clase BusSchool, de la complejidad de cada uno de los pasos que el sistema de gobernado del autobús debe realizar para que el vehículo aumente de velocidad, se frene, o se active el intermitente derecho.

Como dijimos anteriormente, la abstracción suele combinarse con la encapsulación, dejando únicamente como públicos los miembros más generales o abstractos.

```java
BusSchool busSchool = new BusSchool();
busSchool.acelerar();
busSchool.frenar();
busSchool.activarIntermitenteDerecho();
busSchool.acelerarFase01();
```

The method acelerarFase01() from the type BusSchool is not visible

1 quick fix available:

Change visibility of 'acelerarFase01()' to 'package'

Press 'F2' for focus

Los cuatro pilares de la POO: **Polimorfismo**

La palabra **polimorfismo** está formada con raíces griegas y significa "cualidad de tener muchas formas".

Sus componentes léxicos son: polys (muchos) y morfo (formas), más el sufijo -ismo (actividad, sistema) .

En POO, este concepto se refiere a la posibilidad de que objetos de distintas clases, pero que tengan una base común, puedan ser usados indistintamente.

Siguiendo con el ejemplo de la escuela, vamos a suponer que hay dos tipos de profesores, los profesores normales (ProfesoresNormales) y los profesores tutores (ProfesoresTutores), ambos descendientes (heredan propiedades y métodos) de la clase Profesor.

Los cuatro pilares de la POO: **Polimorfismo**

```java
class Profesor extends Persona {

    String titulacion;

    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        return sueldoBase - (sueldoBase * (porcentajeRetencion / 100));
    }
}
```

```java
class ProfesorNormal extends Profesor {

    @Override
    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        float baseIRPF = sueldoBase + 100;
        return baseIRPF - (baseIRPF * (porcentajeRetencion / 100));
    }

}
```

```java
class ProfesorTutor extends Profesor {

    @Override
    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        float baseIRPF = sueldoBase + 200;
        return baseIRPF - (baseIRPF * (porcentajeRetencion / 100));
    }
}
```

Ya hemos codificado las clases **derivadas** ProfesorNormal y ProfesorTutor, que **heredan** de la clase **base** Profesor, que a su vez, tal y como vimos en unas diapositivas anteriores, **hereda** de la clase **base** Persona.

Notad que la clase Profesor es **derivada** de Persona, y a la vez es **base** para ProfesorNormal y ProfesorTutor.

Para entender el polimorfismo, deberemos tener en cuenta que una instancia de la clase ProfesorTutor, a través del mecanismo de la herencia, se considera que es de ese tipo, pero también es de tipo Profesor, y de tipo Persona.

Es decir, Maria, que es tutora, también es una profesora y, por su puesto, es una persona.

Los cuatro pilares de la POO: **Polimorfismo**

```
class Profesor extends Persona {

    String titulacion;

    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        return sueldoBase - (sueldoBase * (porcentajeRetencion / 100));
    }
}
```

```
class ProfesorNormal extends Profesor {

    @Override
    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        float baseIRPF = sueldoBase + 100;
        return baseIRPF - (baseIRPF * (porcentajeRetencion / 100));
    }

}
```

```
class ProfesorTutor extends Profesor {

    @Override
    float calculaSueldo(float sueldoBase, float porcentajeRetencion) {
        float baseIRPF = sueldoBase + 200;
        return baseIRPF - (baseIRPF * (porcentajeRetencion / 100));
    }
}
```

Siguiendo con el código, vemos que la clase Profesor tiene un método calculaSueldo(…), que realiza un cálculo en función del sueldo base y el porcentaje de retención.

Y también vemos, que ese mismo método, está sobre escrito en sus clases **derivadas**, dando lugar a cálculos diferentes.

En el caso de ProfesorNormal, antes de restar la retención, se sumarán 100€ el sueldo base.

Y en el caso del ProfesorTutor, en lugar de 100€, al sueldo base se le sumaran 200€.

Por tanto, podemos considerar que el método calculaSueldo(..) es **polimórfico**.

Los cuatro pilares de la POO: **Polimorfismo**

```java
public class Test {

    public static void main(String[] args) {

        Profesor profesor01 = new ProfesorTutor();
        profesor01.nombre = "Esther";
        float sueldoProfesor01 = profesor01.calculaSueldo(1000, 10);
        System.out.println("El sueldo de " + profesor01.nombre + " es de " + sueldoProfesor01);

        Profesor profesor02 = new ProfesorNormal();
        profesor02.nombre = "Alberto";
        float sueldoProfesor02 = profesor02.calculaSueldo(1000, 10);
        System.out.println("El sueldo de " + profesor02.nombre + " es de " + sueldoProfesor02);

    }

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Test [Java Application] C:\Users\joan.salvador.gordi\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_18.0.2.v20220903-1139\jre\bin\javaw.exe (21 feb 2
El sueldo de Esther es de 1080.0
El sueldo de Alberto es de 990.0
```

Finalmente, podemos comprobar como funciona el **polimorfismo**:

La variable profesor01, que es de tipo Profesor, está instanciada como ProfesorTutor.

La variable profesor02, que también es de tipo Profesor, está instanciada como ProfesorNormal.

Por tanto, ambas serán tratadas de manera común como Profesor, pero al llamar al método calculaSueldo(…), se aplicará el polimorfismo, y para cada una de estas instancias se aplicará el algoritmo de cálculo que tienen codificado en sus respectivas clases.

Muchas gracias